Introduction - CAP Course

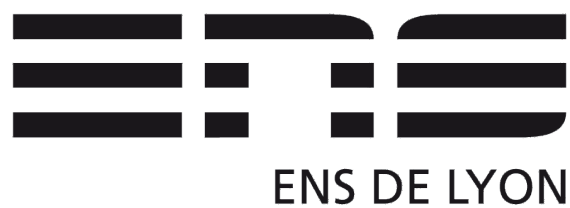
Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

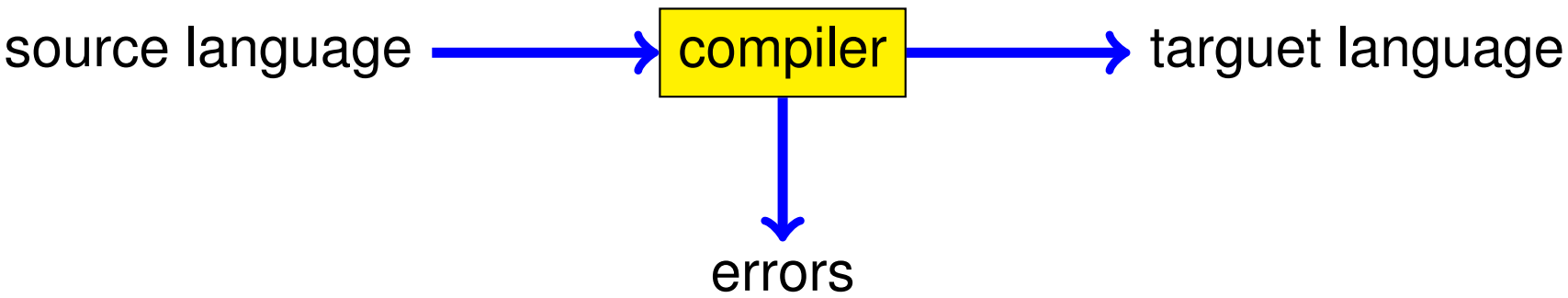
sept 2017



Credits

A large part of the compilation part of this course is inspired by the Compilation Course of JC Filliâtre at ENS Ulm who kindly offered the source code of his slides.

What's compilation ?



Compilation toward the machine language

We immediatly think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (Pentium, PowerPC...)

```
% gcc -o sum sum.c
```

```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);
}
```

$\longrightarrow$

```
00100111101111011111111111000001010111110111111000000000010100
10101111010010000000000010000010101111010010100000000100100
10101111010000000000000001100010101111010000000000000011100
10001111010111000000000011100
```

Target Language

This aspect (compilation into assembly) will be presented in this course, but we will do more :

Compilation is not (only) code generation

A large number of compilation techniques are not linked to assembly code production.

Moreover, languages can be

- interpreted (Basic, COBOL, Ruby, Python, etc.)
- compiled into an intermediate language that will be interpreted (Java, OCaml, Scala, etc.)
- compiled into another high level language (or the same !)
- compiled “on the fly” (or just on time)

Compiler/ Interpreter

- A compiler translates a program P intro a program Q such that for all entry x, the output $Q(x)$ is the same as $P(x)$.

$$\forall P \; \exists Q \; \forall x ...$$

- An interpreter is a program that, given a program P and an entry x, computes the output of $P(x)$:

$$\forall P \; \forall x \; \exists s ...$$

Compiler vs Interpreter

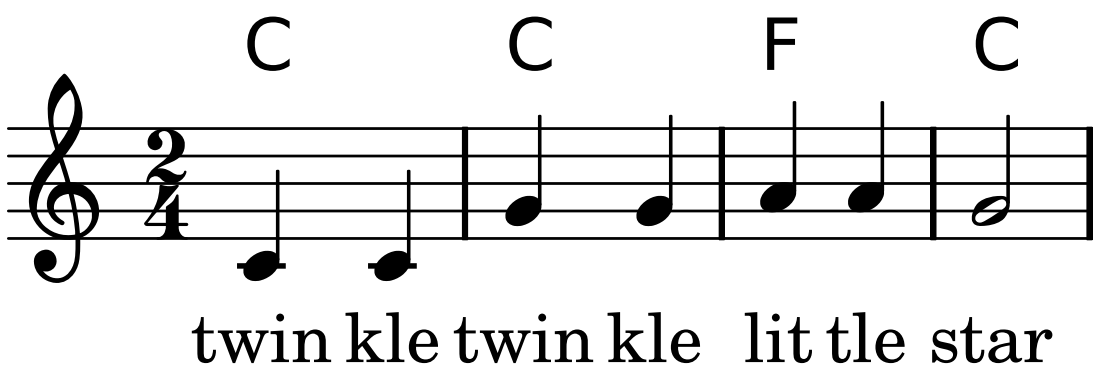
Or :

- The compler makes a complex work once, to produce a code for whatever entry.
- An interpreter makes a simpler job, but on every entry.

- In general the code after compilation is more efficient.

Example

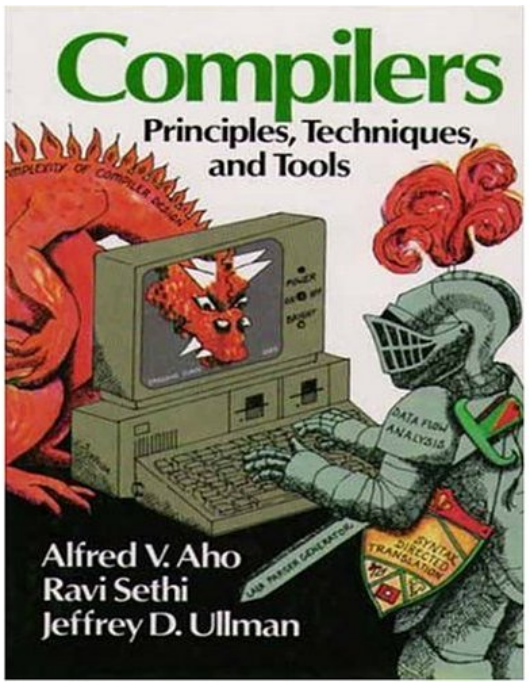

```
\chords { c2 c f2 c }
\new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
\new Lyrics \lyricmode { twin4 kle twin kle lit tle star2 }
```



Compiler Quality

Quality criteria ?

- correctness
- efficiency of the generated code
- its own efficiency



”Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free ! Thus, the most important objective in writing a compiler is that it is correct.”

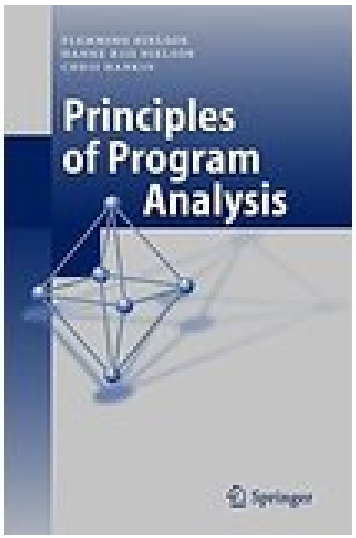
(Dragon Book, 2006)

Program Analysis

To prove :

- Correctness of compilers/optimisations phases.
- Correctnes of programs : invariants

. . . the second part of the course.



Course Objective

Be familiar with the mecanisms inside a (simple) compiler. Be familiar with basis of program analysis.

- And understand the links between them !

Course Content - Compilation Part

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation.
- Code Optimisation.

Lab : a complete compiler for the LEIA architecture !

Support language : Python 3

Frontend infrastructure : ANTLR 4.

Course Content - Analysis Part

- Concrete semantics
- Abstract Interpretation
- A bit of verification : abstract interpretation, Hoare logic, . . .

Labs : abstract interpretation. Support language : Python / Ocaml

Course Organization

- 13 + 1 course slots : Laure Gonnord.
- 14 lab slots : Aurore Alcolei and Valentin Lorentz

The official URL :

`http://laure.gonnord.org/pro/teaching/capM1.html`

Evaluation

- One partial exam.
- Labs.
- A final exam.

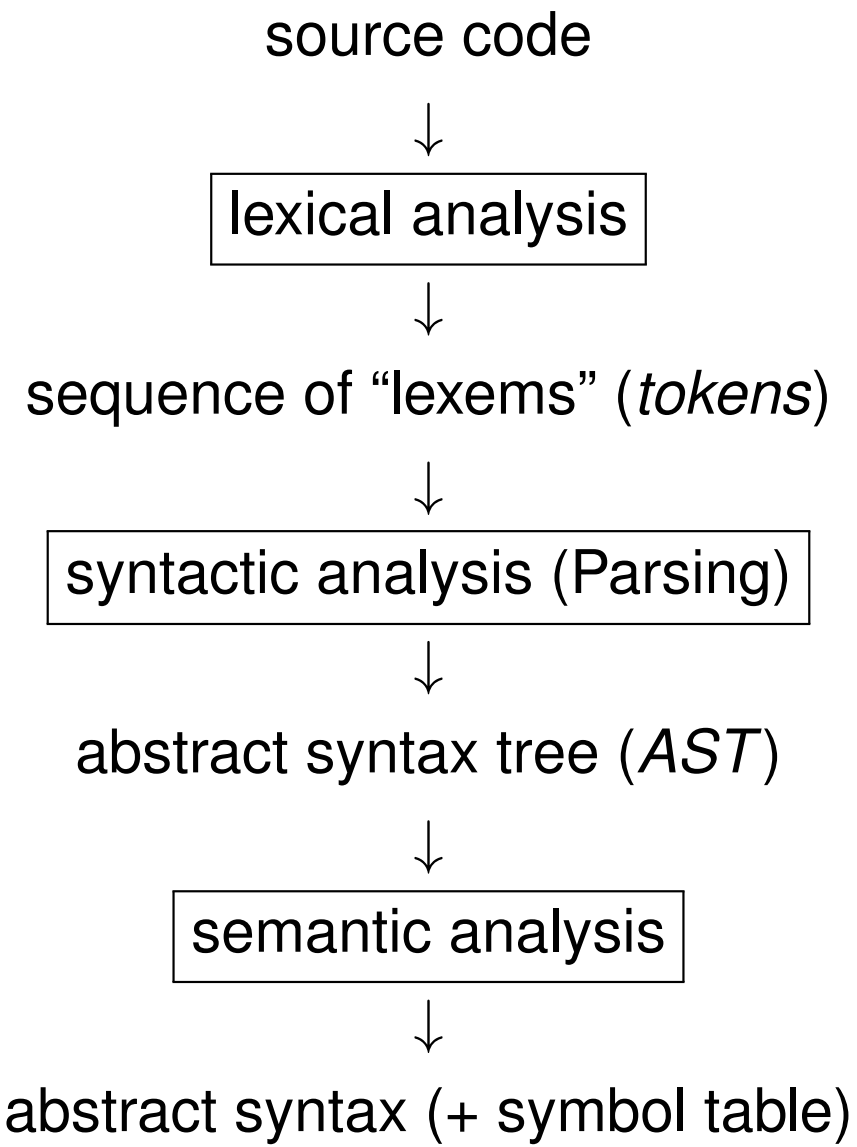
1 Compiler phases

Compiler phases

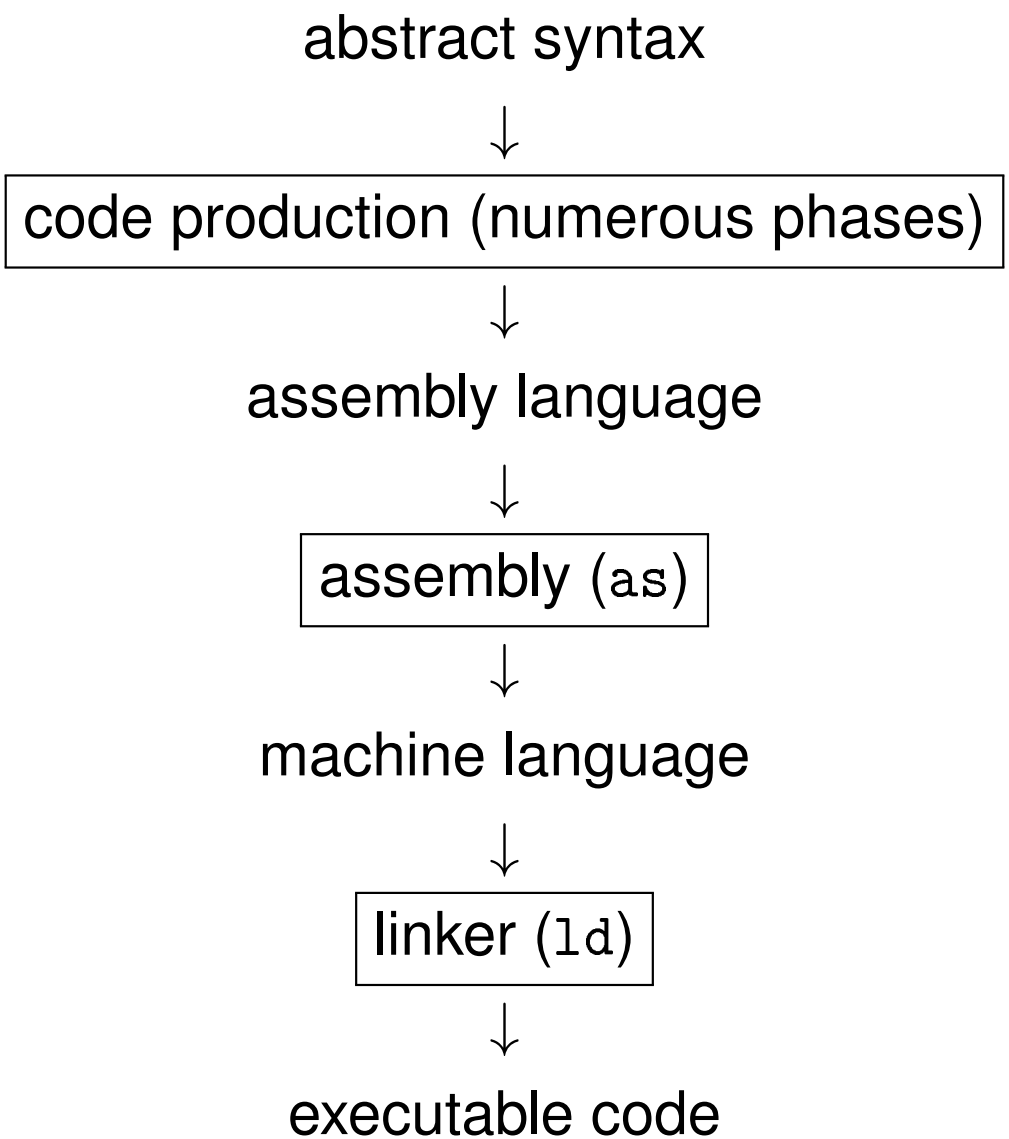
Usually, we distinguish two parts in the design of a compiler :

- an *analysis phase* :
  - recognizes the program to translate and its meaning.
  - launch errors (syntax, scope, types . . .)
- Then a *synthesis phase* :
  - produces a target file.
  - sometimes optmises.

Analysis Phase



Synthesis Phase



Today

assembly